An Introduction to Metaprogramming in Ruby

Marc Paterno

Neat Topics for Programmers

21 December 2010

"I always knew one day Smalltalk
would replace Java.
I just didn't know it would be called Ruby."

— Kent Beck,
creator of Extreme Programming and
Test Driven Development

Goals

- Learn enough Ruby to follow examples
- Become familiar with the Ruby object model
- Become familiar with the meaning and purpose of metaprogramming

What is metaprogramming

Definition

Metaprogramming is writing code that manipulates language constructs (*e.g.* classes and methods) at runtime.

A quick introduction to Ruby

Ruby is dynamic

- Variables do not need to be declared; they "spring into existence" when first encountered.
- Variables do not have types
- Variables are references to objects—objects have "types"[1].

Try this now!

```
>> a = 1          # => 1
>> a.class()      # => Fixnum
>> a = 'hello'    # => 'hello'
>> a.class()      # => String
```

[1] More precisely, objects belong to classes.

Ruby is object-oriented

- Everything is an object.
  - object = state + behavior
- Every object is an instance of a class.
  - behavior comes from classes
- Call `class` to get the class of an object.

Try this now!

```
>> 1.class()              # => Fixnum
>> 'hello'.class()        # => String
>> nil.class()            # => NilClass
>> nil.class().class()    # => Class
```

Classes and objects

class a category of objects that share common functionality

object an instance of a class, with unique state and identity

Every object can be asked for:

- its identity: `x.object_id()`
- its class: `x.class()`

Classes and objects

class a category of objects that share common functionality but not necessarily the same state space

object an instance of a class, with unique state and identity

Every object can be asked for:

- its identity: `x.object_id()`
- its class: `x.class()`

In Ruby, different instances of the same class do not necessarily have the same member data (instance variables)

Invoking behavior

- Behavior is always invoked by sending a message to an object
- Usually this is done with the dot `.` operator
- No free function: sending message to `self`

Try this now!

```
>> i = 1          # => 1
>> i.succ()       # => 2
>> puts(i)        # => nil
1
```

- There are also more exotic ways to send messages

Try this now!

```
>> i.send(:succ) # => 2
```

Writing methods

Functions are called methods.

Ruby code

```ruby
def greeting(name) # def'n starts with def
  "Hello, " + name
end                # def'n ends with end

puts(greeting("Marc"))
puts(greeting("class"))
```

Writing methods

Functions are called methods.

Ruby code

```ruby
def greeting(name) # def'n starts with def
  "Hello, " + name
end                # def'n ends with end

puts greeting "Marc"
puts greeting "class"
```

- Ruby does not require parentheses for method calls

Writing methods

Functions are called methods.

Ruby code

```ruby
def greeting name  # def'n starts with def
  "Hello, " + name
end                # def'n ends with end

puts greeting "Marc"
puts greeting "class"
```

- Ruby does not require parentheses for method calls
- Ruby doesn't even require parentheses in method definitions

Writing methods

Functions are called methods.

Ruby code

```ruby
def greeting(name) # def'n starts with def
  "Hello, " + name
end                # def'n ends with end

puts greeting "Marc"
puts greeting "class"
```

- Ruby does not require parentheses for method calls
- Ruby doesn't even require parentheses in method definitions
- . . . but don't do that, because it is weird

Writing classes

Ruby code

```ruby
require 'date'
class Person
  def initialize(name, dob)
    @name, @dob = name, Date.parse(dob)
  end
  # approximate age in years
  def age
    ((Date.today-@dob)/365).to_i
  end
end
emp = Person.new "Gaius Julius Caesar",
                 "13 July 100 BC"
puts emp.age # => 2110
```

The Ruby Object Model

The critical concepts

- objects
- self
- current class

Inside an Object

- Each object is associated with another object, called its class
  - the associated object's class is always `Class`
  - the associated object is where Ruby starts to look for methods
  - if the required method is not found,
  - method lookup goes "up the chain" to any `included` modules, and if the method is still not found,
  - continues with the class's superclass

Let's look at an example . . .

Method lookup

```
x = 'cat'
```

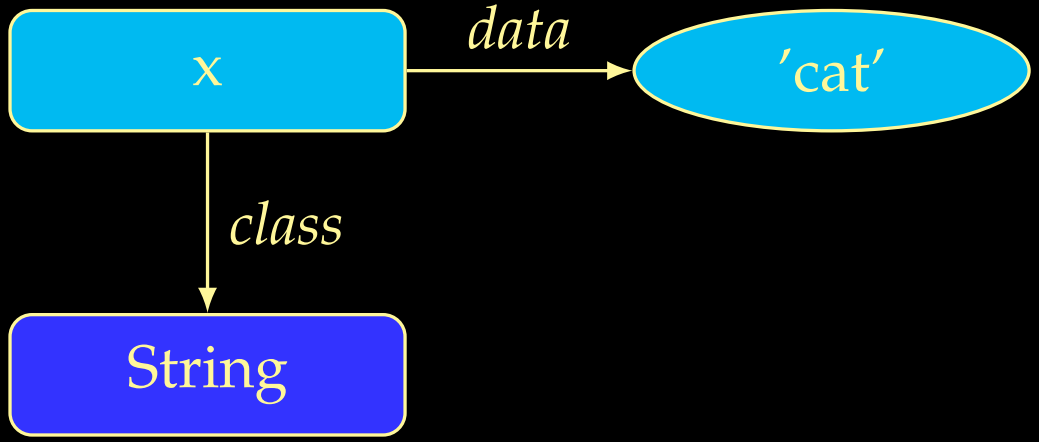
x is the name of an object

Method lookup

```
x = 'cat'
x.class
```

x is the name of an object

x's class is the object named *String*



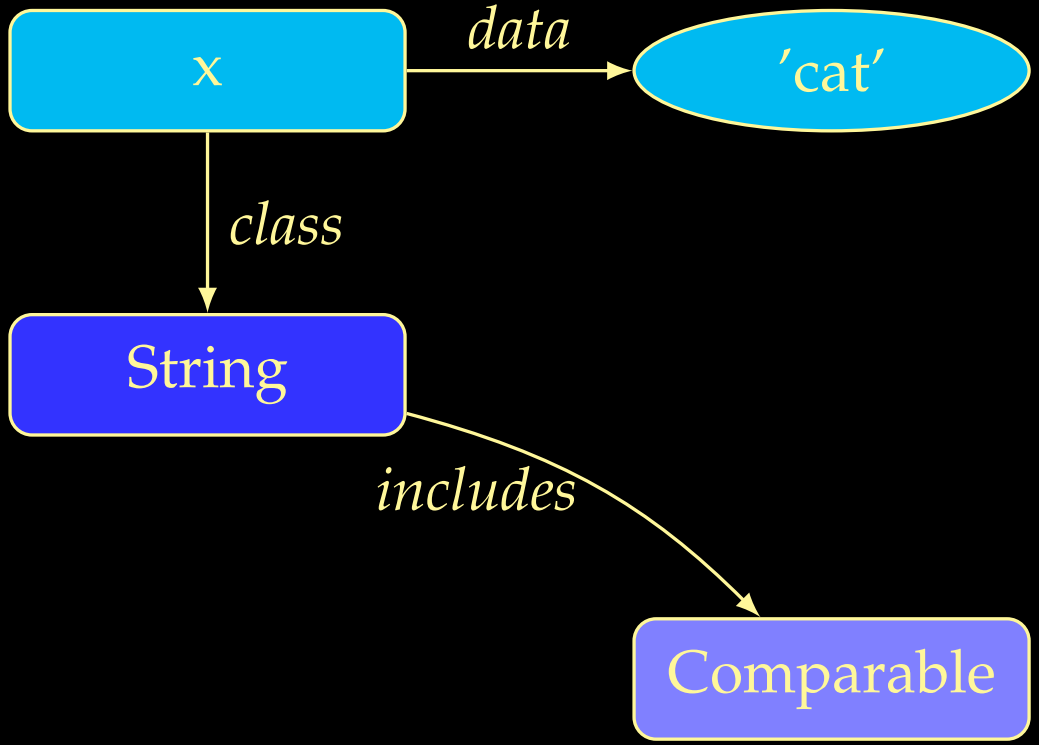
Method lookup

```
x = 'cat'
x.class
x.class.ancestors
```

x is the name of an object

x's class is the object named *String*

String's ancestors are classes and modules



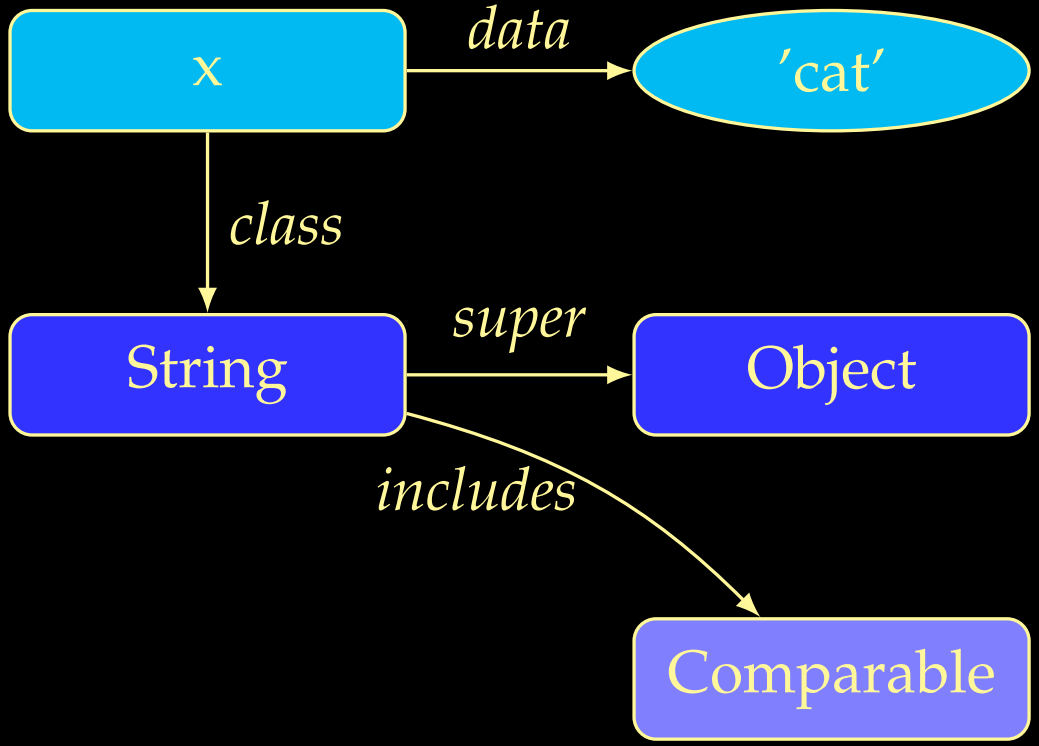
Method lookup

```
x = 'cat'
x.class
x.class.ancestors
```

x is the name of an object

x's class is the object named *String*

String's ancestors are classes and modules



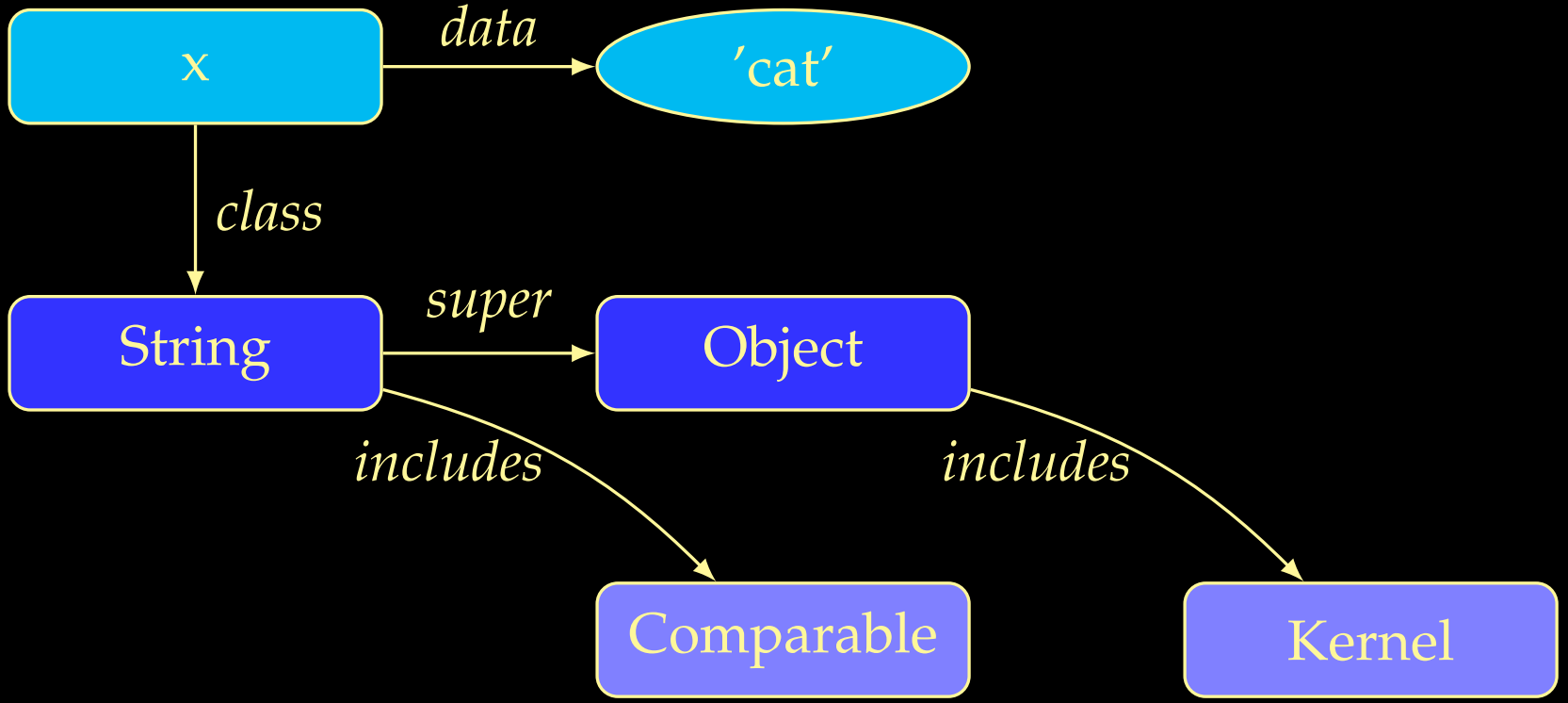
Method lookup

```
x = 'cat'
x.class
x.class.ancestors
```

x is the name of an object

x's class is the object named *String*

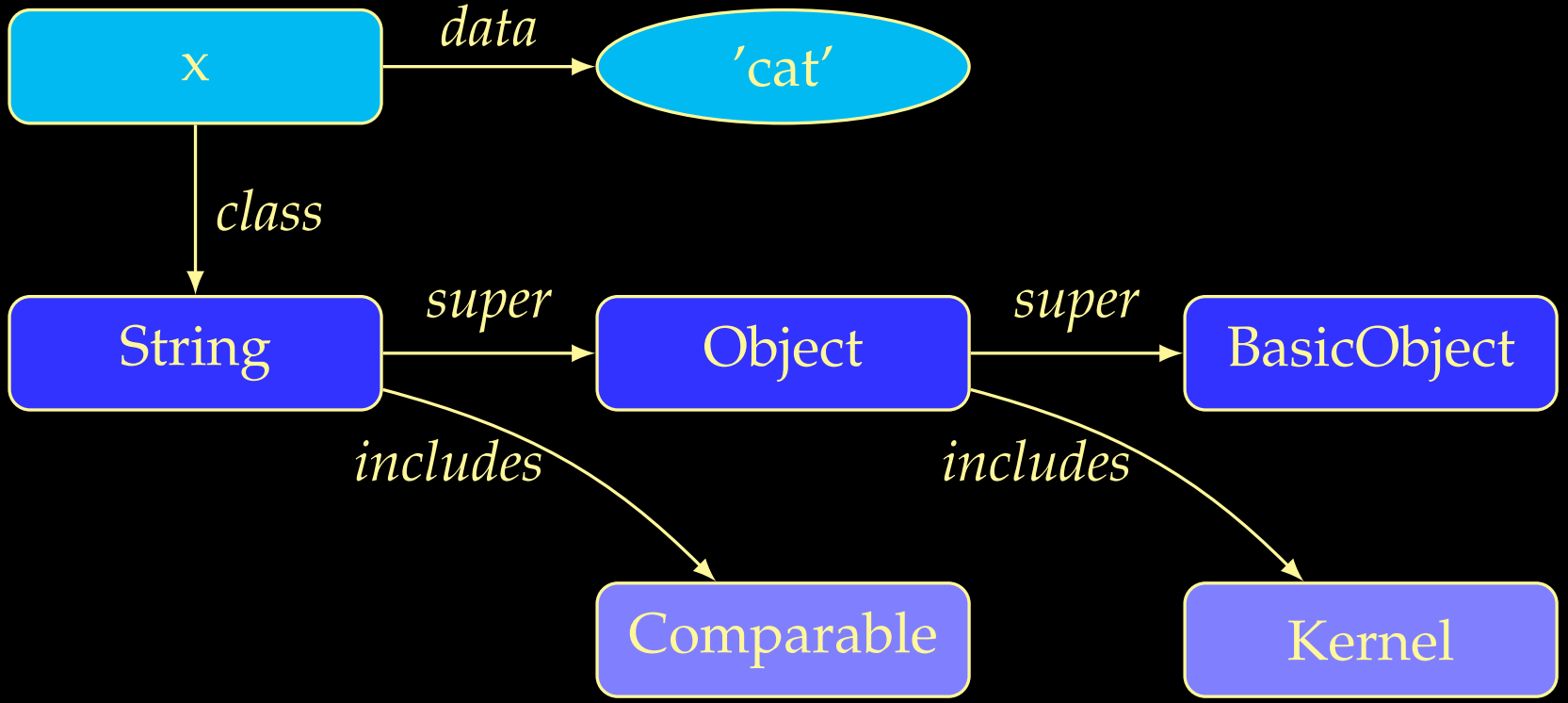
String's ancestors are classes and modules



Method lookup

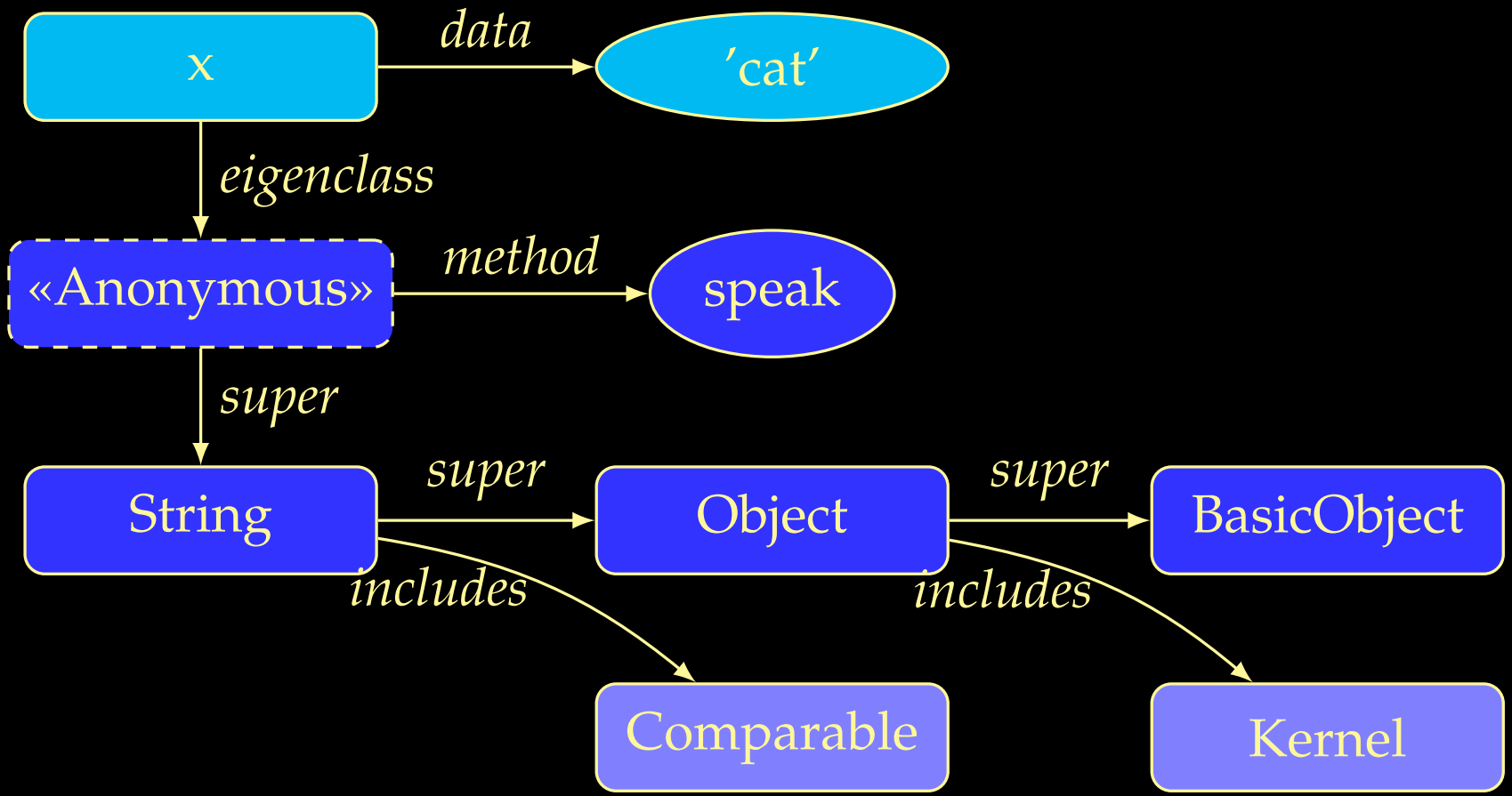
```
x = 'cat'
x.class
x.class.ancestors
```

x is the name of an object
x's class is the object named *String*
String's ancestors are classes
and modules



Method lookup: with eigenclass

```
x = 'cow'
def x.speak
  puts "moo"
end
```



Ruby book

- **Metaprogramming Ruby**, Paolo Perrotta